Software Engineering Notes

Core principles, development methodologies, system design, clean code, testing, and modern engineering practices — from junior to senior mindset.

v2.0 · Updated April 2026

What is Software Engineering?

Software Engineering is the disciplined application of engineering principles to the design, development, testing, deployment, and maintenance of software systems. It goes beyond just coding — it involves process, collaboration, quality assurance, and lifecycle management.

Systematic approach
Requirements → Design → Implementation → Validation → Evolution

Team & scale
Engineering is about building reliable software in teams, for long-term maintainability.

Software Development Life Cycle (SDLC) & Methodologies

📌 SDLC Phases

- Requirement analysis
- System design (high-level & detailed)
- Implementation (coding)
- Testing (unit, integration, system)
- Deployment & release
- Maintenance & evolution

🔄 Popular Methodologies

- **Agile (Scrum, Kanban)** — iterative, customer feedback, adaptive planning
- **Waterfall** — sequential phases, suited for well-defined requirements
- **DevOps** — CI/CD, collaboration between dev & ops, automation
- **Extreme Programming (XP)** — test-driven development, pair programming

💡 Agile + DevOps is the industry standard for modern software teams (2026)

📦 Foundational Principles: SOLID, DRY, KISS & YAGNI

Single Responsibility — one class, one purpose

Open/Closed — open for extension, closed for modification

Liskov Substitution — subtypes must be substitutable for base types

Interface Segregation — avoid fat interfaces

Dependency Inversion — depend on abstractions, not concretions

DRY — Don't Repeat Yourself

> **KISS** — Keep It Simple, Stupid

> **YAGNI** — You Aren't Gonna Need It (avoid over-engineering)

🧹 Clean Code & Engineering Best Practices

- ✔ Meaningful variable/function names (self-documenting)
- ✔ Functions should be small and do one thing
- ✔ Consistent formatting & linting (Prettier, ESLint, Black)
- ✔ Write comments only for “why”, not “what”
- ✔ Favor composition over inheritance
- ✔ Avoid magic numbers; use named constants
- ✔ Error handling: fail fast and explicitly
- ✔ Refactor early, refactor often
- ✔ Code reviews are non-negotiable
- ✔ Keep a healthy test suite (unit + integration)

```
 // Example: clean function (single responsibility)
function calculateTotal(items) {
  const subtotal = items.reduce((sum, item) ⇒ sum + item.price, 0);
  return applyTax(subtotal);
}

function applyTax(amount) {
  const TAX_RATE = 0.08;
  return amount * (1 + TAX_RATE);
}
```

🧪 Software Testing Pyramid & Strategies

Testing Pyramid (Mike Cohn):

- **Unit tests** — many, fast, isolated
- **Integration tests** — fewer, verify modules interaction
- **E2E tests** — few, slow, simulate real user

Other critical tests:

- Regression tests
- Performance / load tests
- Security testing (SAST/DAST)
- Acceptance testing (ATDD)

```
 // Example unit test (Jest - JavaScript)
test('calculateTotal applies tax correctly', () ⇒ {
  const items = [{ price: 100 }, { price: 50 }];
  expect(calculateTotal(items)).toBeCloseTo(162); // 150 + 8% tax
});
```

System Design & Architecture Patterns

🏛️ Common Architectural Patterns

- Monolith (simple start)
- Microservices (independent deployability)
- Event-Driven Architecture (EDA)
- Layered (n-tier) architecture
- Hexagonal (ports & adapters)

📦 Design Patterns (Gang of Four)

- **Creational:** Singleton, Factory, Builder
- **Structural:** Adapter, Decorator, Facade
- **Behavioral:** Observer, Strategy, Command

🔥 Microservices + API-first design dominate 2026 architectures, but monoliths remain valid for many products.

Version Control & Git Workflows

Git is the industry standard. Essential workflows:

- **Git Flow** — feature branches, develop, release, main
- **GitHub Flow** — simpler, main + feature branches
- **Trunk-based development** — short-lived branches, continuous integration

- ✅ Atomic commits & meaningful messages
- Pull request / merge request reviews
- 🚀 CI/CD pipelines run on every push

```
# Git best practices
git commit -m "feat(auth): add OAuth2 login handler"
git push origin feature/oauth2
# then open pull request for review
```

∞ CI/CD & DevOps Essentials

Continuous Integration (CI) — automatically build and test each code change.
Continuous Delivery/Deployment (CD) — automated release pipelines.

🔧 Key Practices

- Automated testing in pipeline
- Infrastructure as Code (Terraform, CloudFormation)
- Containerization (Docker) & orchestration (K8s)
- Monitoring & observability (Prometheus, Grafana)

📈 Popular Tools

- Jenkins, GitLab CI, GitHub Actions
- ArgoCD (GitOps)
- SonarQube (code quality)

📝 Documentation, Technical Debt & Refactoring

Living documentation > heavy outdated docs

- README, architecture decision records (ADR)
- API docs (Swagger/OpenAPI)
- Code comments only when necessary

Technical Debt — shortcuts that accumulate interest. Manage by:

- Refactoring sprints
- Code metrics & linters
- Prioritize debt in backlog

> "Leave the code better than you found it" — Boy Scout Rule

Engineering Soft Skills & Mindset

Communication: articulate trade-offs, write RFCs, lead technical discussions.

Estimation: break tasks down, include buffer for unknowns.

Problem decomposition: divide complex problems into manageable pieces.

Continuous learning: tech evolves rapidly — adopt a growth mindset.

Quick Reference: Languages & Paradigms

Paradigm	Languages	When to use
Object-oriented	Java, C#, Python, C++	Large systems with state and behavior encapsulation
Functional	Haskell, Scala, Elixir, F#	Data transformations, concurrency, immutability
Procedural	C, Go, Rust (multi)	Performance-critical, system-level
Scripting & dynamic	Python, JavaScript, Ruby	Rapid prototyping, glue code, web

Software Engineering Trends (2026)

- **AI-assisted development** — GitHub Copilot, Cursor, LLM-based code generation
- **Platform engineering** — internal developer portals (Backstage)
- **Serverless & edge computing** — reduced ops burden
- **Security by design (DevSecOps)** — shift-left security
- **Rust adoption** — memory safety without GC
- **WASM (WebAssembly)** — polyglot deployment
- **Green software engineering** — energy-efficient code

Recommended Learning Resources

The Pragmatic Programmer Clean Code (Robert Martin)

Designing Data-Intensive Applications

System Design Interview (ByteByteGo)

Refactoring Guru (design patterns)

Google SRE Book

Great engineers ship code that solves real problems, but great **software engineers** build systems that last and evolve.
